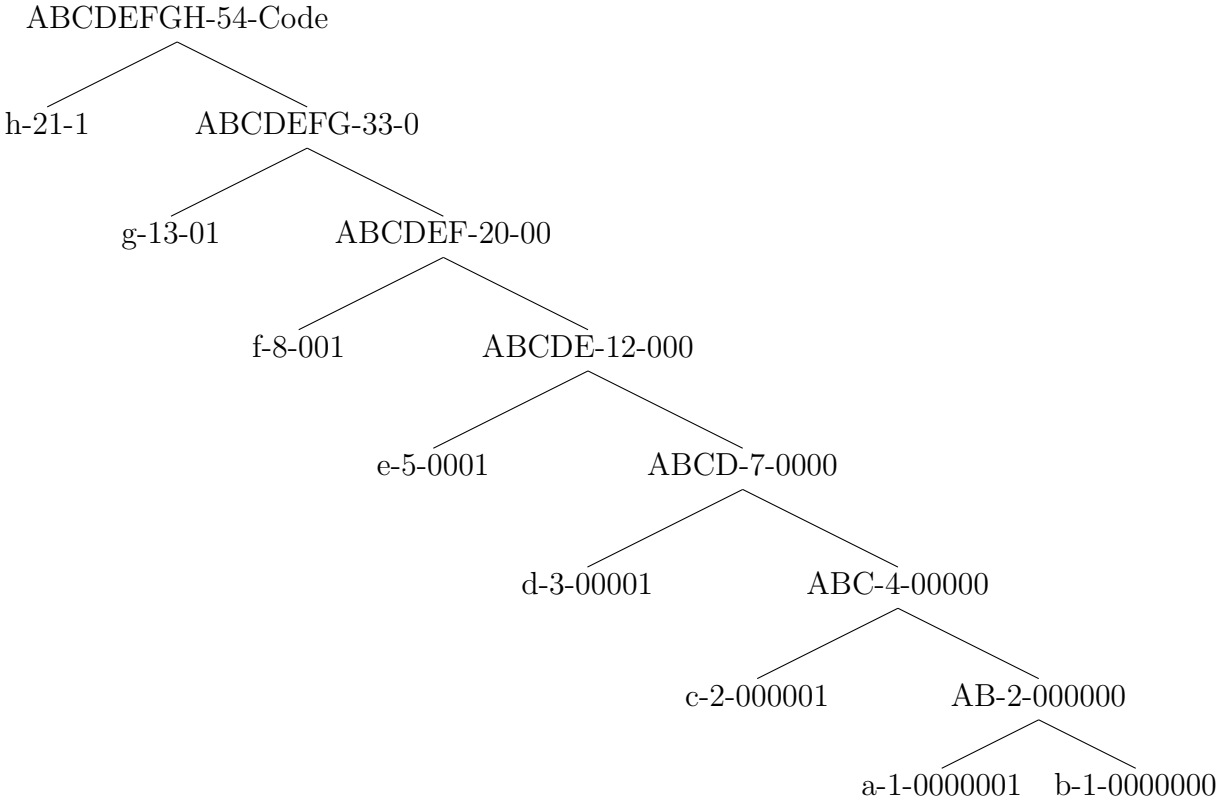
CSCI 3104 Spring 2014 Problem Set 4
Andrew Howe (08/25)
Ellie Daw (05/31)

---

1. *Recall that Huffman codes are constructed in a greedy fashion.*

   (a) *What is an optimal Huffman code for the following set of frequencies, based on the First 8 Fibonacci numbers?*



(b) *How many optimal Huffman codes are there for this set of frequencies?*

If we are keeping a structured binary tree where we maintain that the left node always holds the lesser of the two values and the right holds the greater, then there are 8 optimal Huffman codes for this set of frequencies. We are able to interchange nodes a and b, a node above that we can interchange nodes AB and c this gives us four unique codes. Furthermore by changing the numbering of the nodes to 0's on the left node and 1's on the right node we have 8 optimal codes.

If we have a tree that randomly assigns the nodes to left or right regardless of frequency we have 2^n optimal codes where n is the number of nodes that have two children, in this case 128 optimal configurations.

(c) *Generalize your answer to Find an optimal code when the frequencies are the First n Fibonacci numbers*

Given my answer in part (a) we can generalize this to a recursive function that assigns the last fibonacci number the number 1 and then recursively calls itself adding a prefix of 0 to each recursive call such that the $n-1$ fibonacci number is assigned the code 01 and the $n-2$ is 001, we keep recursing until there are only two numbers that remain then the recursion stops and the first fibonacci number is assigned a code with $(n-2)0$'s ending in a 1 and the second fibonacci number is assigned a code that consists of $(n-1)0$'s

2. *Professor Hagrid is struggling with the problem of making change for n cents using the smallest number of coins.*

   (a) *Give a greedy algorithm, that takes $O(n)$ time, to make change consisting of quarters (worth 25 cents), dimes (10 cents), nickels (5 cents) and pennies (1 cent). Prove that your algorithm yields an optimal solution.*

   A greedy algorithm that works in $O(1)$ time is given by the Psuedocode below

```
MakeChange(double amount)
{
 int newAmount=(amount*100).toInteger();
 int quarters=newAmount/25;
 newAmount=newAmount%25;
 int dimes=newAmount/10;
 newAmount=newAmount%10;
 int nickels=newAmount/5;
 newAmount=newAmount%5;
 return (quarters,dimes,nickels,newAmount);
}
```

   This algorithm always yields the optimal solution because given the nature of the U.S. currency system or any other currency with change in these denominations it is always optimal to follow the greedy-choice and take as many coins of the largest denomination you can before moving onto lower denominations. See (c) for my proof, that when applied to this currency set shows that it contains the greedy-choice property.

   (b) *Prove that the greedy algorithm always yields an optimal solution in this case.*

   The greedy algorithm always yields an optimal solution in this case because each denomination is a factor of c greater than the previous so it is obvious for any l it is always optimal to take one coin worth c^l rather than taking c number of coins worth c^{l-1}. This proves that the greedy-choice property exists for a currency in these denominations.

(c) *Give a set of coin denominations for which the greedy algorithm does not yield an optimal solution, and explain why*

A currency that comes in the denominations of 1,10,15 does not have the greedy-choice property we can prove this by searching all cases under the sum of two times the largest denomination for contradictions because any value greater it would be optimal to take as many coins of the largest denomination we can. By inspection we quickly see that the greedy algorithm fails when we are making change for 20 cents. The greedy algorithm would yield one 15 cent piece and five pennies while the optimal solution is two dimes. This set of denominations does not have the greedy-choice property because we have found a contradiction where the algorithm does not produce the optimal result.

3. *Give a O(lg n)-time algorithm which finds the median of A union B and prove that it is correct.*

To find an $O(lg(n))$ algorithm this means that the algorithm must cut the amount of items in half on each recursion. An algorithm that satisfies these requirements is one that compares A[A.length/2-1] and B[B.length/2 -1], the lower median value in each array or the middle value in odd length arrays. It then keeps the second half of the array (the part containing higher values) of the array that had the lower median value and the first half of the array that had the higher median value untill we have only one item in each array the median value is then given by the lower of these two values. This method ensures that we are never discarding the lower median of the union of these two arrays and that the lower value of the last two items will always be the median value as defined in the book.